EN.605.202.81: Introduction to Data Structures

Peter Sullivan

Lab 1 Analysis

Due Date: 7/2/24

Lab 1 Analysis

Data Structures

The lab assignment requested that we set up a process for maneuvering from a prefix expression to a postfix expression. There are many ways that this could be implemented. For this lab I created three classes. The Expression class enables users to convert from Postfix to Prefix and Infix, and Prefix to Postfix and Infix. The expressions class utilizes two additional classes. The first is the operand class which combines strings together based on the desired input and output. For example, going from a prefix to a postfix expression. The final class is the stack array class. This class utilizes the python list data structure and creates a stack object that mimics the methods of a stack abstract data type. For example, we use the list built in methods to create methods like isempty(), pop(), push(), and peek(). These methods allow us to loop through the prefix strings and appropriately keep track and combine operands and operators together when converting to postfix.

Data Structure Justification/Implementation

Before choosing data types and structures, I first needed to understand how to convert from prefix to postfix. When converting from prefix to post fix, the first step is to read in the expression from right to left. This really means we're reading in the string in reverse. While reading in the string we are keeping track of operands and keeping track of operators. When an operator comes up, this is when we take two of the most recent items that were processed, and we join those into a new operand.

So, we need to read in the data in reverse, and then we need to keep track of recent items. A stack data structure is a perfect data structure for this application. First reversing a string is quite easy with a stack. You go through each element and push onto the stack, once the stack is full, the last item on the stack is now the first item to be popped off. Now you can pop the whole stack and you have a reversed string. For the process of converting from prefix to postfix, we push onto the stack and pop twice whenever we receive an operator. We then combine the operator with the operands to create a new operand and push back on the stack or list. This is also a very great application for a stack. Since we are looking for a last in first out application. We continue this process until there are no more characters in the prefix expression. As we process and combine each character, we should end up with only one single item in the stack, which should be the postfix representation of the initial prefix expression.

A stack is a very easy way to grab the most recent items. No indexing or slicing is required, so this is a very good solution to this problem. Another option would be using a deque, though we would have to make sure that we are grabbing the most recent instead of the first in first out method that a deque offers.

Another option is to use a recursive solution. This would be an acceptable way to process the prefix expression. To implement a recursive solution, we would have the function start parsing through the prefix and slowly start creating the postfix. While parsing, whenever we combine to create a new operand such as a +b. We could then send over the prefix unprocessed, and the postfix processed to the function to be called again. We would iteratively keep combining until the base case where the prefix string has been completely processed and the postfix string is now complete.

Complexity

Recursive Implementation
The recursive solution would methodically process through the string which would give us a O(N) complexity. We would then call the function multiple times until we process the prefix string and create the postfix string, so the complexity would be O(N)*M. M would be the amount of calls required for the base case to be reached. As the expression gets very long the M can be ignored and the complexity would become O(N). The space complexity would also be O(N) *M as we would keep creating space every time the program is run, but that would also move to O(N) as the number of calls would be a constant based on the size of the input string.

Stack Implementation
The expression class is doing most of the work for my program. If we are looking at prefix to post fix method in the expressions.py file, I first initialize an empty stack (complexity O(1)). I then loop through the input expression in reverse. To read in reverse, I use the reverse order method that I created. This method implements a stack to reverse the order of the string. In this method, I have two loops that loads the stack and unloads the stack. The complexity of reverse order would be O(N)*2. I then have another for loop, where I load up the stack for each item. I then either push or pop twice. If I pop twice from the stack, I then initialize an operand object. I then push the operand object to the stack using the operand combine object. All of these are O(1) complexity. Assuming that we can convert to a postfix, I then return the popped stack. Here is an outline of the complexity assuming the worst case:

- Initiate stack
    - stack = StackArray()
    - O(1)
- Reverse String:
    - reverse_order(self,string):
    - O(N)*2
- Loop through reverse string:
    - For loop
    - O(N)
- Pop twice
    - pop2 = stack.pop()
    - O(1)+O(1)
- Initiate Operand Object
    - operand = Operands(pop1,pop2,char,'Pre_Post')
    - O(1)
- Push operand combined
    - stack.push(operand.combine_operands())
    - O(1)+O(1)
- Return pop
    - Return stack.pop()

    - 0(1)

The complexity would be O(N)*2 +O(N) +O(7). This would turn into O(N) complexity. The space complexity would also be the same as it would only be as large as the input string provided O(N).

Both the recursive and iterative process would have a complexity of O(N) for space and time.

If I had to choose between a stack solution or a recursive solution. I would probably go with the recursive solution due to its simplicity. We are only doing one combine operand operation at a time, which is very easy to keep track of. Both solutions work and have a similar space and time complexity, so I would choose the simpler coding option, which would be the recursive solution.

Enhancements

The original lab asked to only convert from prefix to postfix. For this lab I added additional functionality to convert from Prefix to Postfix, Prefix to Infix, Postfix to Prefix, Postfix to Infix. To minimize user input, I added in helper functions with the expression class to pick the correct method when running. For example, the user would only need to enter Postfix Infix, and the program will automatically pick the correct method to perform that option.

I also added in two additional arguments that the user can enter if desired from the command line. The default if the two additional are not entered is to run from prefix to postfix. But the user has the option to add two additional arguments at the end of the cmd line, input_type and output_type. The following works:

- python -m lab1 resources\Required_Input.txt output/output.txt
- python -m lab1 resources\Required_Input.txt output/output.txt Prefix Postfix
- python -m lab1 resources\Required_Input.txt output/output.txt Postfix Prefix
- python -m lab1 resources\Required_Input.txt output/output.txt Prefix Infix
- python -m lab1 resources\Required_Input.txt output/output.txt Postfix Infix

The two arguments can be adjusted based on the desired output and the input files provided. Depending on the input type and output type selected, this will tell the expression object which method to run in order to correctly parse and convert from the original input type to the desired output type.

Lessons Learned
If I were going to do this project again, I would spend much more time on the test cases/edge cases. I think it would have been a lot easier for me if I had spent more time trying to understand the method of how the prefix is processed to prefix instead of trying to make the code work.

I've never used the arg_parser before, and I am very happy I've learned this package. It's incredibly useful to be able to create a package or program that can be run by the command line. I also found the pathlib package very useful. I found at the very last minute the functionality of having optional and default arguments via argparse, which is incredibly useful!

I had a difficult time making the project into a module. I had first created the process and had it running via a juptyer notebook, which made it easier for testing. For future labs, I want to create test cases at the beginning. I then want to start the project as a module. That way for testing purposes, I can utilize the command line to process inputs and start debugging immediately.

I learned the hard way that project versioning is important. I made changes to the project and then suddenly nothing was working. After many control Z's, I was then able to run the module again. As instructed by the professor, I utilized git and GitHub for versioning on the lab. I started to work on my lab in smaller pieces, creating a branch for each enhancement, and pushes those changes into the master branch after testing. This made me feel a lot more comfortable about trying changes, when I knew that I had a working version saved in the master branch.

I also think I took on a little too much for me to work on. Next time I will focus just on the project before trying to pull in enhancements right away.